Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Juha Siren

Prototyping to Integrate Requirements Engineering and Software Architecture Design

Master's Thesis
Espoo, November 15, 2016

Supervisor: Professor Marjo Kauppinen, Aalto University
Advisor: Varvana Myllärniemi D.Sc. (Tech.)
Jarko Uzal

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Juha Siren

Title:
Prototyping to Integrate Requirements Engineering and Software Architecture Design

Date: November 15, 2016 **Pages:** 58

Major: Software Engineering and Business **Code:** T3003

Supervisor: Professor Marjo Kauppinen

Advisor: Varvana Myllärniemi D.Sc. (Tech.)
Jarko Uzal

Requirements engineering (RE) and software architecture design (SAD) are common challenges in complex software projects. Several studies indicate that requirements engineers and software architects should work closely together. However, the integration of RE and SAD can be challenging in practice. Prototyping is one possible solution to this challenge, however there is very little research done on prototyping in this context.

The goal of the study is to find out how prototyping can support the integration of requirements engineering and software architecture design.

This study was done using the case study methodology and by a literature review. The case study was conducted in a small Finnish software company that applied prototyping for the development of a new software product. The purpose of the prototyping project was to find the requirements for the minimum viable product and to verify the technical feasibility of our solution.

Prototyping helped us understand customer requirements better. This was achieved by using user interface (UI) mock-ups and by demoing the implemented prototypes to customers in order to get feedback. Prototyping also helped us define and find new architecturally significant requirements (ASR) during the implementation of the prototypes and customer demos. Furthermore, prototyping was used in software architecture design and evaluation. Both exploratory and experimental architectural prototypes were used to evaluate the architectural impact of various architectural design decisions. Architectural prototyping was found to be beneficial as a learning tool, and the knowledge gained from prototyping could be used to make informed architectural design decisions.

The findings of this study indicate that prototyping can support the integration of RE and SAD by collaboratively defining ASRs. Furthermore, requirements engineers and software architects should work together while preparing for customer demos, as various prototypes can be used to support the discussion with customers.

Keywords: prototyping, software architecture, requirements engineering, architecturally significant requirements

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Juha Siren

Työn nimi:
Prototypointi vaatimustenhallinnan ja ohjelmistoarkkitehtuurisuunnittelun integroinnissa

Päiväys:	15. marraskuuta 2016	**Sivumäärä:**	58
Pääaine:	Ohjelmistotuotanto ja liiketoiminta	**Koodi:**	T3003
Valvoja:	Professori Marko Kauppinen		
Ohjaaja:	Tekniikan tohtori Varvana Myllärniemi Jarko Uzal		

Vaatimustenhallinta ja arkkitehtuurisuunnittelu ovat yleisiä haasteita ohjelmistoprojekteissa. Usean tutkimuksen mukaan vaatimusten määrittelijöiden ja ohjelmistoarkkitehtien pitäisi tehdä tiivistä yhteistyötä. Näiden osa-alueiden integraatio on kuitenkin havaittu haasteelliseksi käytännössä. Prototypointi on yksi mahdollinen ratkaisu, tosin prototypointia on tutkittu erittäin vähän tässä kontekstissa.

Tämän työn tavoitteena on selvittää, miten prototypointi voi tukea vaatimustenhallinnan ja ohjelmistoarkkitehtuurisuunnittelun integrontia.

Tutkimusmenetelminä käytettiin tapaustutkimusta sekä kirjallisuuskatsausta. Empiiriset havainnot perustuvat pienessä suomalaisessa ohjelmistoyrityksessä tehtyyn projektiin, jossa kehitettiin uutta tuotetta prototypoinnin avulla. Projektin tavoitteina oli varmistaa tuotteen tekninen toteutettavuus sekä selvittää, mitkä vaatimukset on vähintään toteutettava, jotta asiakkaat voisivat ottaa tuotteen käyttöönsä.

Prototypointi auttoi asiakkaiden vaatimuksien ymmärtämisessä. Tämä onnistui prototypoimalla käyttöliittymää sekä esittämällä toteutettuja prototyyppejä asiakkaille ja keräämällä palautetta. Lisäksi prototyyppien rakentaminen ja niiden esittäminen asiakkaille auttoivat arkkitehtuurisesti merkittävien vaatimusten määrittelemisessä ja uusien vaatimusten löytämisessä. Prototyypit tukivat myös arkkitehtuurisuunnittelua ja arkkitehtuurin arviointia. Prototyyppien avulla voitiin selvittää, millainen vaikutus erilaisilla arkkitehtuurisilla päätöksillä on arkkitehtuuriin. Prototypoinnilla saavutettiin myös parempi ymmärrys arkkitehtuurista, minkä seurauksena pystyttiin tekemään parempia arkkitehtuurisia päätöksiä.

Tämän työn tulokset osoittavat, että arkkitehtuurisesti merkittävien vaatimusten määrittely prototypoinnilla kannustaa vaatimustenmäärittelijien ja ohjelmistoarkkitehtien välistä yhteistyötä. Lisäksi havaittiin, että vaatimustenhallinnan ja ohjelmistoarkkitehtuurisuunnittelun integraatiota tukee prototyyppien valmistelu asiakasdemoja varten.

Asiasanat:	prototypointi, ohjelmistoarkkitehtuuri, vaatimusten määrittely, arkkitehtuurisesti merkittävät vaatimukset
Kieli:	Englanti

Contents

Chapter 1

Introduction

1.1 Background

There are various challenges in the software development industry related to requirements engineering and software architecture design. Requirements engineering (RE) struggles with rapidly changing, poorly structured, vague and conflicting requirements [25, 29, 38]. Quality requirements and other architecturally significant requirements (ASRs) that are not discovered in the early phases of a project can lead to software architects making the wrong architectural decisions, resulting in a lot of rework and refactoring later on to fix the software architecture (SA) [25]. This issue is further emphasized as software companies focus more and more on quickly shipping a minimum viable product without paying much attention to proper architectural design and documentation [22].

Furthermore, it has been found that understanding the dependencies between requirements and software architecture is essential in order to build successful and cost-effective software [10]. There are many benefits of requirements engineers and software architects working closely together: better understanding of the technical feasibility of requirements, ability to make informed architectural decisions, discovering hidden requirements and improved requirements negotiations with customers [2, 8, 14].

Although there has been a lot of research on the benefits of integrating requirements engineering and software architecture design (SAD), there are still challenges related to this integration in the industry. The most fundamental challenge is the mismatch between the information that requirements specifications contain and the information that software architects require [11]. Software architecture design is mostly driven by quality requirements and other ASRs, while requirements engineers often focus heavily on visible

features [8, 11].

Therefore ASRs can be thought of as the link between RE and SAD. In this study we evaluate the use of prototyping to support the integration of RE and SAD by defining ASRs. The existing literature on prototyping has strongly focused on RE and SAD separately, so there is very little existing research on using prototypes in this context.

1.2 Research problem and questions

The goal of the study is to find out how prototyping can support the integration of requirements engineering and software architecture design. The following research questions describe the main focus of the study:

1. How does prototyping support understanding customer requirements?
2. How does prototyping support defining the architecturally significant requirements?
3. How does prototyping support the design and evaluation of software architecture?

1.3 Thesis structure

The research methodology used in this study is presented in section two. Previous work related to prototyping in software engineering is reviewed and analyzed in section three. The data, observations and other results from the empirical part of the study are presented in section four. In section five the results are discussed in detail: the research questions are answered based on the results, the results are compared to the previous work described in section three to determine if any new results were found, and the validity and reliability of the results are evaluated. Section six concludes and summarizes the most significant parts of this study and provides answers to the research problem.

Figure 1.1 describes in which subsections each research question is discussed from the literature review and case study perspectives.

Figure 1.1: Thesis structure: Sections related to each research question

Chapter 2

Research Method

2.1 Case description

The study was carried out in a small Finnish software company, who is bringing a new payroll solution to the market. To better compete in the market, the company is also planning on developing two additional software products that it will integrate and bundle together with the main product. These two products are workforce scheduling as well as time tracking. With these three products bundled, the company can offer its customers an all-in-one package with out-of-the-box integrations, so that customers no longer need to configure and maintain multiple systems from multiple vendors for the single business process.

In the project we developed prototypes for the scheduling and time tracking products. The project was market-driven, meaning that there was no single customer, but instead an opportunity in the market had been identified and the new product will be a solution to an existing problem across many potential customers. During the project, the prototypes were shown to multiple potential customers to get important feedback during the project, and the feedback was used to create improved versions of the prototypes.

2.2 Research process

The empirical research for this paper was done using the case study methodology [28]. Figure 2.1 describes the research process used in this study. The research process phases are described in more detail in sections 2.2.1 and 2.2.2.

In the beginning of the research process, we defined the research questions and made the necessary preparations before the product development started.

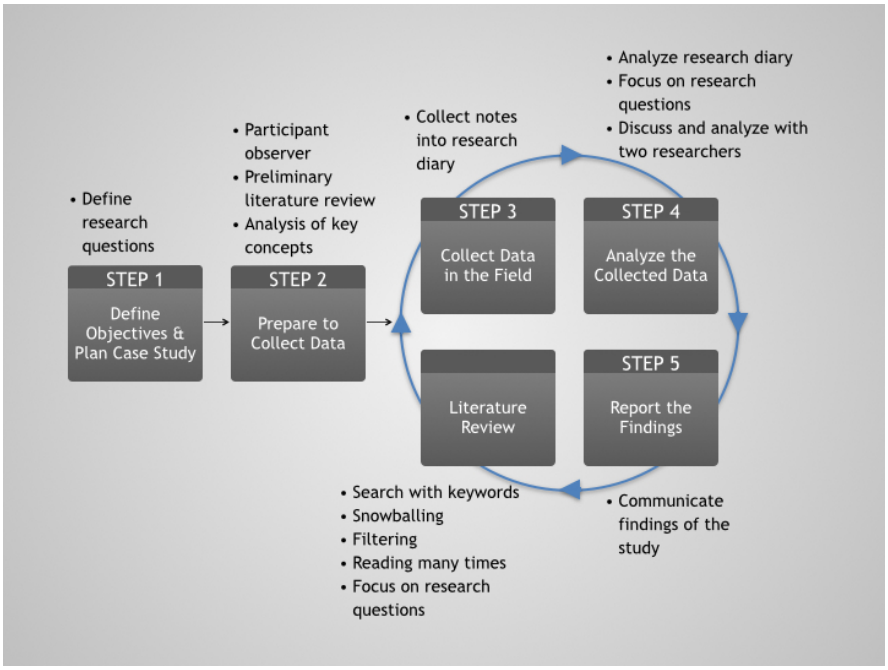

Figure 2.1: Our Case Study Research Process

These preparations included a preliminary literature review and analysis of key concepts, to become familiar with the prototyping process. After these two preparatory steps, the rest of the process was iterative. In each iteration, new data was collected and analyzed, the findings were reported and the literature review was continued to reflect upon the new findings.

2.2.1 Empirical research process

Case study was selected as the research method because we were doing an exploratory study and gathering qualitative data, and we wanted to keep the design of the study flexible and carry the study out iteratively and incrementally. These are the very characteristics of case studies [28], which made it the best option for this research.

A case study consists of five steps, as described by Runeson & Höst [28]:

1. Define objectives and plan the case study
2. Prepare to collect the data
3. Collect data in the field

4. Analyze the collected data

5. Report the findings

In the first step, you plan the case study by defining several elements: defining the objective, the case, the theory, research questions, research method and data selection strategy [27]. Having planned all of that, we had a clear vision of what we would focus on during the study. However we did stay flexible with the plans, and they were slightly modified during the study, but the original main focus did remain the same.

The second step is to prepare to collect the data, i.e. evaluate and select the data gathering techniques that will be used in the case study. In this study, we chose to gather qualitative data using the participant observation method [30]. The benefit of using participant observation is to find phenomena that might be missed with other research methods where the researcher is less involved [30]. In this study, the researcher was part of the team and was involved in all of the activities. Furthermore, in preparation for the data gathering we also performed a preliminary literature review in the second step. After the preliminary literature review, we analyzed the key concepts to become more familiar with the existing research. Thus we could decide e.g. which types of prototypes should be created, and also learned the requirements engineering and software architecture design related activities and their purpose.

The third step is to collect the data. In this study, we decided to collect the data into a research diary, where all meetings and informal interactions within the team or with the customer were logged. Each entry in the research diary contained all the participants, the duration of the meeting or discussion, the topics discussed and the decisions made, if any. This step was done iteratively and in parallel with the literature review described in section 2.2.2. The data gathering was strongly supported with the preliminary literature review that was carried out in step two. We were able to focus on the key concepts during the data gathering, as we understood them better due to the literature review.

Fourth step is to evaluate and analyze the data. The main objective is to derive conclusions from the data while keeping a clear chain of evidence, meaning that the reader can see how the results and conclusions are derived from the collected data [30]. Furthermore, we analyzed the data iteratively and in parallel with data collection, as is typical in case studies [28].

In this study, the data analysis was done in several steps, starting from the research diary and from analysing the existing literature to reflect upon our own findings. First, the notes from the research diary were combined

into narrative descriptions of what happened during the project, organized chronologically and structured based on what activity was involved. Second, these narratives were then analyzed, focusing on the research questions to capture relevant ideas and findings for this study. After that, the findings were discussed together with the professors supervising this study, who helped find new perspectives and ask new questions regarding the data and its analysis. Through these discussions we were able to identify high level findings and to ensure that all the relevant findings are presented in this paper. Furthermore, through these discussions we noticed several findings that hadn't been explicitly written down in the research diary.

The final step is to prepare the report and write it. The purpose of the report is to communicate the findings of the study. It can also be used to judge the quality of the study [28]. In this study, the report was written iteratively and in parallel with both the data collection and the analysis steps.

2.2.2 Literature review process

The literature review was carried out by first searching for a large number of potentially relevant material for this study. These papers were then quickly evaluated to filter out the papers that were not relevant enough for the study. This was done by browsing the abstract and conclusions as well as some figures and tables. These two steps were done multiple times during the literature review process with different keywords to find new material. The keywords that yielded the most important sources for this paper were "prototyp* requirements engineering" and "prototyp* software architect*". Furthermore, the snowballing method was used, i.e. studying the relevant references of the papers to find more detailed information about specific topics.

The relevant papers were then analysed in depth to find and understand them well. Some papers didn't end up containing a lot of critical information and were only read once, while others were extremely valuable for this study and were read and analysed many times over. When reading for the first time, relevant sentences and paragraphs were highlighted with different colors based on what topic or research question it was related to. This was done in order so that finding the important sections of each paper would be easier later on.

Chapter 3

Previous Work

3.1 Requirements engineering

Requirements engineering in the software development environment faces several challenges. Software development can often start from poorly structured and vague requirements, with a variety of stakeholders having conflicting requirements [38]. The user requirements change quickly and often during development, while fast releases are being expected. Starting from ambiguous project goals, it is a huge challenge to be able to elicit and validate these frequently changing requirements. [38].

At the same time, fast-changing technology and increased competition and software complexity have increased the pressure on the development process for software companies [20]. Therefore effective requirements engineering is increasingly important for software companies to keep pace with the increasing difficulty. The key to addressing the issue of rapidly changing requirements is to identify the core requirements of a software system [25]. This is because the core requirements describe the stakeholders' goals that will probably persist for the longest time and will probably lead to a software architecture that can accommodate changes in other requirements [25].

Furthermore, requirements are often difficult to define in advance. Defining requirements in advance can also result in requirements that are obsolete and to software that ultimately doesn't meet customer needs [21, 29]. In addition, defining everything in advance is usually not feasible due to limited time and resources for RE activities [36]. Furthermore, requirements often emerge only after significant analysis of models or prototypes and after users provide feedback on the models or prototypes [25]. Therefore requirements engineering should be conducted continuously throughout a software project instead of as a one-off in the start of a project [29].

Damian and Chisan have shown in their study [13] that improvements in RE practices have a positive effect on developer productivity, software quality and risk management. In addition, it also helps in other software development activities such as testing, peer-reviewing and project tracking.

One of the main reasons for software project failure is poorly executed requirements engineering: poorly organized or expressed requirements, unrealistic and unnecessary requirements, requirements that are changing too fast, scope creep and uncontrolled change in project goals [15, 20]. Therefore it is widely acknowledged that effective requirements engineering is important to avoid project failure.

Another major challenge in requirements engineering is that quality requirements are often neglected [10, 21]. The main emphasis is often on visible features, while quality concerns are being ignored [10, 21]. This can result in developers delivering solutions that don't meet the customer's quality expectations [10].

However, in practice software vendors tend to simply gather requirements from their clients and believe that requirements engineering is now done [29]. This is clearly insufficient, however, and instead software companies should strive for codevelopment with the client to establish an on-going requirements engineering process [29]. This can significantly help with known RE challenges such as insufficient quality communication with the customer and not understanding the big picture, which often causes misunderstandings with the customer resulting in future rework and increased costs [21].

3.1.1 Requirement classification

Requirements can be categorised into three classes: functional, quality attribute and constraints [4]. Functional requirements state what the system must do and how it needs to react to e.g. specific user input [4]. Quality attribute requirements are qualifications for the functional requirements or the overall system [4]. Functional requirement qualifications can state e.g. how fast a specific function needs to be or how robust the system is in error situations, while e.g. time to deploy or operational cost items are qualifications for the overall system [4]. Constraints are design decisions with zero degrees of freedom, e.g. a design decision such as the programming language has already been made due to external factors such as insufficient resources to train the developers to learn a new language [4].

3.1.2 RE activities

The requirements engineering process is generally split into the following four steps: elicitation, analysis, documentation and verification [15].

3.1.2.1 Elicitation

Elicitation is the process of discovering, extracting or somehow else figuring out the needs of the customers, users and other stakeholders [18].

According to Maiden and Rugg [23], when developing a new system, rapid prototyping and scenario analysis are considered the best suited techniques to elicit the information from stakeholders. The reason these are considered best is that they simulate the required system and its interaction with the environment, making it is easier for the stakeholders to understand it and compare it with other systems [23]. Other techniques include semi-structured interviews and brainstorming, which can be used in conjunction with prototyping to further enhance the discussion [15, 23]. A single technique, however, cannot provide a complete solution to a complex problem, so therefore multiple techniuqes should be combined to achieve the best results [23].

3.1.2.2 Analysis

In the analysis phase of the requirements engineering process, the information elicited from the stakeholders is analysed to increase understanding and to search for incompleteness, inconsistency and contradictions between the requirements [18]. Furthermore, the requirements are prioritized to determine which requirements should be included into a specific release. Green also noted that in modern markets there are often many competing products and it is the differentiating capabilities of each system that capture additional market share [15]. Thus, prioritization should also take into account these nice-to-have features, which are important to differentiate from competitors [15].

3.1.2.3 Documentation

Requirement documentation means that the requirements are recorded and distributed in some suitable way to different stakeholders. For example the developers want a detailed low-level description of the requirements, while the end-user is only interested in the system-level requirements.

One important purpose of documentation is to enable the verification of requirements, so that there is a measure by which we can determine the success and/or completion of the project.

3.1.2.4 Verification

In the verification step, the documented requirements are delivered to the stakeholders for approval. If it is rejected, the conflicts need to be resolved so the process continues in the elicitation phase to determine why it was not accepted. When all stakeholders are in agreement, the requirements can be frozen. Green notes that from this point onward it is important to track the requirements for any changes and additions, which must be approved again by all stakeholders, in order to prevent too much scope creep as well as to keep the project goals unchanged [15].

3.2 Software architecture design

A system's architecture is the set of principal design decisions made during its development and any subsequent evolution [35]. The production and development of high-quality, successful software products is dependent upon these principal decisions [24]. The software architecture has a significant impact on the quality attributes that the system can achieve, and the architecture can allow or prevent almost all of the quality attributes of the system [12]. Shaw defines software architecture as the nature of interactions among the components [31]. Software architecture is a high abstraction level description for the system's overall organization, decomposition into simpler components, the separation of functionality between these components and the interactions between them [16, 31].

Software architecture design consists of three activities: architectural analysis, architectural synthesis and architectural evaluation [19]. The purpose of architectural analysis is to define the problems which the architecture must solve [19]. During architectural analysis, the architectural concerns and context are examined, filtered and/or reformulated in order to come up with a set of ASRs [19]. Architectural concerns are often expressed as requirements of the system, but they can also be constraints such as mandatory standards or regulatory requirements [19]. Architectural context describes the environment of the system, including things such as business goals and characteristics of the organization [19].

Architectural synthesis is the core of software architecture design [19]. During architectural synthesis, software architects produce candidate architectural solutions that satisfy the ASRs defined in architectural analysis [19]. Essentially, architectural synthesis is the link between the problem and solution space, as the input for architectural synthesis is a set of ASRs and it produces a candidate architectural solution [19]. The candidate architectural

solutions can present alternative and/or partial solutions to the problem [19].

Architectural evaluation ensures that the right architectural design decisions are made [19]. The candidate architectural solutions produced during architectural synthesis are evaluated against the ASRs [19]. The candidate architectural solutions are validated or invalidated, and eventually architectural evaluation produces a validated architecture [19]. A validated architecture consists of a set of candidate architectures which are mutually consistent and also consistent with the ASRs [19].

The three main activities of software architecture design (analysis, synthesis and evaluation) do not proceed sequantially, due to the complexity of the design [19]. Instead, they are performed multiple times iteratively, in no predictable order and at different levels of detail, until the final and validated architecture is achieved [19].

3.3 Architecturally significant requirements

Architecturally significant requirements (ASRs) are those requirements that have a significant impact on software architecture, i.e. they are critical, of high risk, volatile, involve expensive refactoring or have legislative impact [2, 8]. Not all requirements have the same architectural impact, for example the requirement "system should have 99.999 percent availability" has a much larger impact on the software architecture than the requirement "temperature should be displayed in Celsius instead of Fahrenheit" [8].

ASRs have a significant impact on software architecture, which means that it has a high cost of change [8]. In addition, if ASRs are wrong, incomplete, inaccurate or lack details, software architects cannot make informed design decisions and the resulting architecture is also likely to contain errors, which will result in costly rework later on [2, 8]. Therefore it is important to identify the ASRs at the early stages of the software development process [8]. However, all decisions are not made in the beginning of the project and customers aren't sure about their exact needs, which makes it more difficult to define the ASRs in advance [8].

Chen et al. presented descriptions, indicators and heuristics for ASRs to make it easier to identify them [8]. In addition to ASRs being hard to define and articulate, they found that ASRs are often vaguely described, they tend to be initially neglected, hidden within other requirements, variable and situational [8]. Vaguely described ASRs can result in software architects making bad design decisions, because the ASRs are missing some details and therefore the architects need to make assumptions which may or may not be correct [8]. Some requirements initially don't seem to have architectural

significance and are therefore ignored, and they are only identified as ASRs after a high cost has been incurred [8]. ASRs are often embedded and hidden inside other requirements instead of being emphasized in the requirements. Requirements always change over time, and thus also ASRs are variable [8]. And finally ASRs are situational, meaning that a requirement might be architecturally significant in one project while being "just a requirement" in another [8].

ASRs have some indicative characteristics which can be used to distinguish them from other requirements [8]. The first of these indicators is how wide impact the requirement has on the system. ASRs often have a wide impact on the system, affecting multiple components or the entire system as a whole; the wider the impact, the more architecturally significant the requirement is [8]. The second characteristic is that ASRs often target trade-offs, meaning that architects must select a design which compromises some requirements to meet others [8]. The third indicator is strict requirements, which force some architectural decisions because they cannot be satisfied by any alternative design options [8]. Indicator number four is assumptions breaking: architects always need to make at least some fundamental assumptions that might be wrong or the environment changes so that the assumption is no longer true, for example if the business expands to other countries in different time zones, turning down the server at midnight for maintenance is no longer acceptable [8]. The final indicator is that ASRs can be difficult to achieve or technologically challenging, for example achieving good latency is hard if it is not considered early on [8].

All of these indicators require substantial technical knowledge, which many requirements engineers don't have [2, 8]. The following heuristics can guide requirements engineers to ask correct questions to find requirements that are potentially architecturally significant [8]. The first heuristic is quality attributes: a requirement that specifies a software system's quality attributes it is almost always architecturally significant [2, 8]. The second heuristic is that the core features of a software system are likely to be ASRs, as they are part of the fundamental assumptions that the architecture is built upon [8]. This is also closely related to the quality attributes, since the core features usually affect the relevant quality attributes [8]. Third are the requirements that impose constraints on the system, such as financial or time constraints or other constraints affected by existing architecture or customer's technical decisions [8]. The final heuristic is that requirements that define or affect the environment in which the software system will run are often ASRs, since software running in a different environment often has a different architecture.

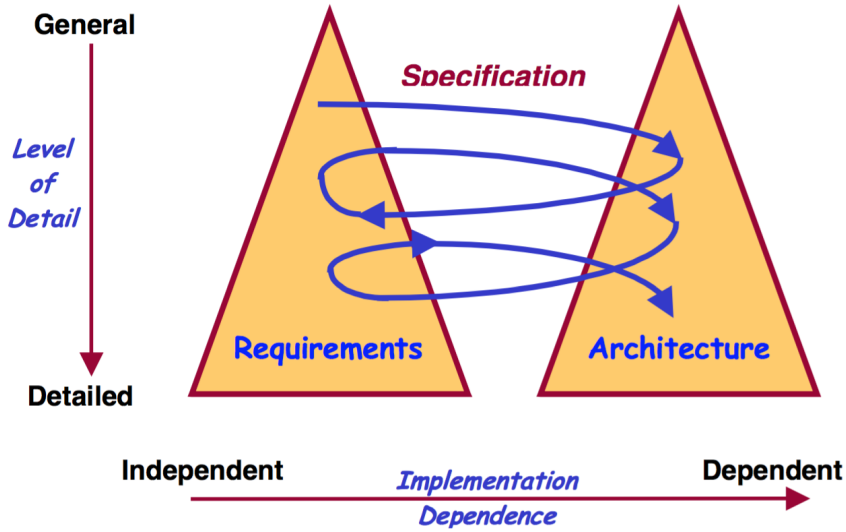

Figure 3.1: Twin Peaks Model [25]

3.4 Integration of RE and SAD

The Twin Peaks Model by Nuseibeh [25] is illustrated in Figure 3.1. It describes the iterative and concurrent process of progressively creating more detailed requirements and design specifications [25]. The twin peaks of requirements and architecture describe the dependencies that exist between requirements and software architecture [25]. It is essential to understand and leverage the dependencies between requirements and software architecture design in order to build successful and cost-effective software [10]. One of the key points of the model is that not only should RE feed information to SAD, but SAD should also feed back into the RE process [25].

Furthermore, Ferrari et al. [14] suggest that RE and SAD processes should be tightly integrated in order to gain critical insight on the technical feasibility of the requirements. This is due to their findings that 35% of all RE decisions are affected by the existing software architecture. In addition, they found that the RE decisions are often affected by details outside of the component being analysed, so a big picture understanding of the full architecture is crucial. Therefore RE agents should work together with software architects during requirements elicitation and negotiation to gain critical insight on the technical feasibility of the requirements [14].

Anish et al. agree that it is necessary to integrate people with different expertise to make informed architectural decisions [2]. Architectural im-

plications are often hidden in the requirements that need to be unearthed, however business analysts lack the technical knowledge to infer and articulate the architectural impact [2, 8].

Chen et al. found that requirements engineers and software architects should work tightly together during the RE process for improved requirements negotiation [8]. For example some architecturally significant requirements don't create sufficient value to justify their cost. Therefore it can be a good idea to try and negotiate with the customer to make the requirement less demanding or to drop it altogether [8]. The software architects can bring this kind of feedback about the requirement impact to the requirements engineers to make better requirements decisions [8].

Shekaran et al. explain key reasons why RE and SAD should be coupled in an interweaving process: better understanding of the system's specifications, evaluating alternative architectures in order to create specifications that can be economically implemented, and easier for developers to move within the bounds of requirements and architecture [32]. Moreover, people often expect that changes which seem simple in the problem space should be also simple and cheap to implement in the solution space. However this might not be the case depending on the software architecture. Therefore software architects can bring valuable information to requirements engineers, who often do not have the knowledge of the system to understand the implications of potential changes in requirements [32].

As we discovered in section 3.3, ASRs are often vaguely described, initially neglected, and that quality attributes are almost always ASRs. On the other hand, as was discussed in section 3.1, one of the main reasons for software project failure is poorly executed requirements engineering, e.g. quality requirements often being neglected during RE resulting in solutions that do not meet the customer's quality expectations. Combining this information, it seems that ill defined ASRs are a major cause of software project failure.

Therefore requirements engineers and software architects need to work closely together in order to clearly define ASRs, and thus be able to create solutions that meet the customer's quality concerns. Cleland-Huang et al. argue that it is important to proactively elicit quality requirements from project stakeholders during early phases of the project to be able to design architectural solutions that satisfy the quality concerns [10]. Early prototypes and architectural evaluations can be used to demonstrate that the delivered system is able to meet its quality goals [10].

3.5 Prototyping

3.5.1 Prototyping definition

A prototype is an initial preliminary version of a solution or a system. Prototyping is the process of designing, building and evaluating prototypes. Prototyping can be done in two ways: throw-away prototypes are discarded after development while evolutionary prototypes evolve and can eventually become part of the final solution [33].

Szekely defines the fast-prototyping concept as creating a small-scale version of a complicated system in order to gain critical knowledge for building the full system [34]. Furthermore the prototype also provides a clear picture to all the stakeholders of what the final system will look like, and also helps with requirements elicitation and verification [38]. Bäumer et al. [5] defines prototyping as developing executable software for experimental purposes in order to improve software project planning and execution. Regardless of the definition, however, the purpose of prototyping is always to gain more insight into designing the final product [21]. Prototyping is usually most beneficial as a process rather than the prototype itself as a product [3]. Prototyping is used as a "learning vehicle" to understand more precisely what the target system should be like [3].

According to Green [15], the first prototype typically represents the minimum viable technology, and the prototype contains only a subset of the initial requirements. The first prototype can be used as a proof-of-concept, to see whether or not the selected technology works as expected, and the success or failure of this prototype will determine the continuation of the project.

This type of prototype can be thought of as a minimum viable product, as the requirements are reduced to contain only the most critical core features [15]. This is very useful, because very many requirements are only needed for the production environment, so the prototype often doesn't need to contain administrative tools and other support features. In addition, some prototypes are never used by the real end-users, but instead are only ever used and demonstrated by the developers. Therefore performance, usability and other quality attributes are not that important for the prototype, so leaving them at a slightly lower level is often recommended for the prototype, even though it might go against some initial requirements.

Boehm presents a prototyping life cycle model, which attempts to solve the need to update the requirements after a period of time, when the problem and the system are better understood [6]. The model is visualised by a spiral, where each 360 degree rotation represents one iteration of the prototype. At the end of each cycle there is a decision point, where the project is evaluated

based on the new information gained during the previous iteration. In the spiral, the distance from the origin describes project costs and each cycle produces a more commercially viable prototype.

3.5.2 UI prototyping

Prototyping was found to be very beneficial for requirements engineering already in the early 90s, when Windsor and Storrs found that using prototypes can reduce the risk of misunderstandings and different interpretations of system specifications [37]. Furthermore it was found that prototyping is cost-effective and possibly the cheapest and most practical way of doing iterative interface design [1, 17]. In addition, prototyping is a very effective way to involve users in an early phase of development and to do user centered design [1, 17, 37].

One of the most significant benefits of UI prototyping is that it is very concrete and comprehensible to users, and you can gain early feedback from users [37]. It is much easier to understand a visual representation than e.g. a written description of a complex system, making prototyping an excellent tool to achieve mutual understanding between developers and non-technical stakeholders [21].

UI prototypes work as unambiguous documentation, solving the issues related to agile projects often having very minimal documentation [21]. As the prototypes are very concrete, they reduce the ambiguity, incompleteness and inconsistency of requirements [21]. Another benefit was found that it is more motivating to update the visual prototype rather than textual documentation, meaning that the visual prototypes are more likely to be up-to-date than other forms of documentation [21].

Furthermore, prototypes are a way to improve the communication between stakeholders [7, 21, 26]. It is often difficult to achieve more communication with stakeholders, so increasing the quality of the communication is one key to improving communication. Prototyping provides a common language that all stakeholders can understand, which helps in reaching mutual understanding faster [21]. In addition, prototyping has been found to result in increased client satisfaction since what clients see is what they will get [38]. Therefore prototyping can reduce the gap between customer expectation and the vision of the developers.

On the other hand, prototyping does not solve all problems related to requirements engineering. One significant challenge with UI prototyping is that it can be challenging to understand the big picture [21]. Focusing on one feature or UI at a time might result in partially optimized solutions, but it is easy to forget or not fully understand the real business needs of

the customer, which can lead to bad decisions that are detrimental for the project as a whole [21]. This problem was further highlighted by focusing too much on end-user satisfaction, instead of taking the time to understand the big picture and making a large-scale plan [21].

Another challenge is that quality and architectural requirements are often neglected because only visible features are being focused on [21]. These architecturally significant requirements might not have any impact in the early phases of development, but will likely require a lot of rewriting and refactoring later on [21].

Furthermore, the fast cycle of creating new prototypes can result in customers having unrealistic expectations about the state of the final product, thinking that it is almost ready [21]. This can also result in more pressure for developers to get the final product ready quickly, and using the prototype as a skeleton for the final product, which was not intended and which can negatively affect the software quality [21].

3.5.3 Architectural prototyping

In order to ensure that a system will meet its most important architectural quality requirements, it is imperative to start building for and evaluating these quality requirements at an early phase of development [3]. For this purpose, architectural prototypes can be built and used for experimentation at a very early stage of development [3]. Bardram et al. defines an architectural prototype as follows: it consists of a set of executables created to investigate architectural qualities related to concerns raised by stakeholders of a system under development [3].

Architectural prototypes are found to be beneficial and cost-effective in demonstrating intrinsic properties of the potential architecture at an early stage of architectural analysis and design [3]. Therefore architectural prototypes can be used to reduce risk and in order to demonstrate important concerns to various stakeholders [3]. For example, an architectural prototype can be used as a skeleton system for the final product, and it can therefore be used as the primary way of communicating the architectural design to the development team [3]. On the other hand, an architectural prototype can demonstrate that the system can achieve the customer's most important quality requirement, e.g. performance [3].

There are five main characteristics of an architectural prototype [3]. First, architectural prototypes are created for exploration and learning of the architectural design space [3]. In contrast to the traditional prototyping definition, architectural prototypes focus strictly on the software architecture itself, largely ignoring the intent of the system under study. This enables

software architects to experiment and learn about e.g. different software architecture patterns at a low cost [3].

Second, architectural prototyping primarily focuses on architectural quality attributes of the target system [3]. One of the software architect's main challenges is to define a software architecture which balances conflicting quality requirements in a satisfactory way. Architectural prototypes offer a way for software architects to build, observe and measure how this balance can be achieved.

Third, architectural prototypes often do not contain functionality itself, but are instead skeleton system that facilitate functionality and address non-functional architectural qualities of the target system [3]. However, Christensen and Hansen discovered in their empirical investigation [9] that less than half of the software architects and developers agree to this characteristic, and almost all architectural prototypes contained at least some domain or end-user functionality.

Fourth, architectural prototypes often address architectural risk [3]. Architectural risk is partly related to the second characteristic, as being able to find a proper balance between conflicting quality requirements can be a risk. However, there are also architectural risks related to aspects that are new and unknown to the organization or to the software architects. In these cases, architectural prototypes can be created in order to acquire knowledge e.g. about technologic platforms or programming models [3].

And fifth, architectural prototypes address the issue of knowledge transfer and architectural conformance [3]. This means that the architectural prototype serves as a reference system which demonstrates key architectural decisions, patterns and styles to the development team. Therefore architectural prototypes that demonstrate the desired quality attribute balance are not thrown away, but instead used as the backbone for developing the final product. This also ensures that the designed architecture is identical to the built architecture [3].

Furthermore, architectural prototypes can be classified into exploratory, experimental and evolutionary prototypes [3]. Exploratory architectural prototypes are created in order to explore the architecture design space. Their primary purpose is to facilitate communication and learning among stakeholders of a software architecture [3]. Exploratory prototypes are mainly used when there is a high degree of uncertainty about how to build the target system, when it is extremely beneficial to have a practical demonstration that is grounded in concrete source code instead of speculation [3]. Typically multiple exploratory architectural prototypes are created, analysed and compared with each other in order to find the optimal solution [3].

Experimental architectural prototypes, on the other hand, are created in

order to evaluate a specific architectural decision [3]. Their main purpose is to determine experimentally whether or not the proposed software architecture satisfies the quality requirement [3]. Experimental architectural prototypes are often created by software architects, who want to prove that the system can meet specific quality requirements when it is built according to a specific architecture. This also serves as early proof of conformance to quality requirements for the customer [3]. Furthermore, as there are often trade-offs between various architecture qualities, experimental prototypes can be created in order to find the optimal balance by understanding exactly how the trade-off manifests itself in real code. Usually only a single experimental architectural prototype is created and evaluated [3].

Evolutionary architectural prototypes are created as a series of prototypes, which are built upon each other. This often leads to a skeleton system with no functionality, but can be used as the basis for the development of the final product [3].

Christensen and Hansen found that the distinction between experimental, exploratory and evolutionary architectural prototypes is more of a theoretical separation [9]. In industrial practice, however, most prototypes contained characteristics of both experimental as well as exploratory prototypes, and most prototypes were evolutionary as well [9].

Christensen and Hansen also found that one key property stressed by several software architects is the importance of demonstration to stakeholders such as business decision makers [9]. If decision makers have conflicting or impossible requirements, architectural prototypes can be built to make it very clear to them that they require the impossible [9]. This type of persuasion was found to be a characteristic of several prototypes, meaning that some functionality was built into the architectural prototype in order to demonstrate the issue and to be able to persuade stakeholders into a specific architectural decision [9].

One downside of architectural prototyping compared to other architectural exploration and evaluation techniques is cost [9]. Substantial effort needs to be invested into the architectural prototypes in order to be optimally useful [9]. This issue is further emphasized due to the industry practice of using architectural prototypes also as demonstrators, as this requires more effort to implement [9]. However, attempting to reduce these costs would also reduce the benefits of architectural prototypes, e.g. there would be a risk of real-world characteristics of the scenarios being missed [9].

Chapter 4

Results

4.1 Project overview and goals

The prototyping project was carried out with a very small team. The author of this paper was responsible for the software architecture design and was the only developer involved. In addition, the team included the product owner, and during the project a graphic designer joined the team. The product owner had a strong vision for the product and was actively discussing with potential customers in meetings and at conferences to gain understanding and knowledge of what the customers need.

There were two major goals for the project. The first one was to find out the customer requirements for the solution and to verify that these requirements satisfy the customer's needs. In other words, we wanted to find out what the most important features are for the customer and to find out what the minimum set of requirements would be so that the customer would be able to use the solution in production. The second goal was to verify the technical feasibility of the solution and to learn more about the related technologies in order to select the best technologies to build the products with.

To achieve our goals, we used the process visualised in Figure 4.1 to develop the prototypes. Before the prototyping began, the product owner had gathered the initial requirements for the system. Then, in the beginning of each prototyping iteration, the graphic designer and the author worked together on analysing the requirements and creating a UI design based on the requirements, after which the graphic designer created a mock-up of the UI design with photoshop. The mock-up was then validated with the product owner before the UI design was implemented in the code. After implementation the prototype was then validated either by the product owner

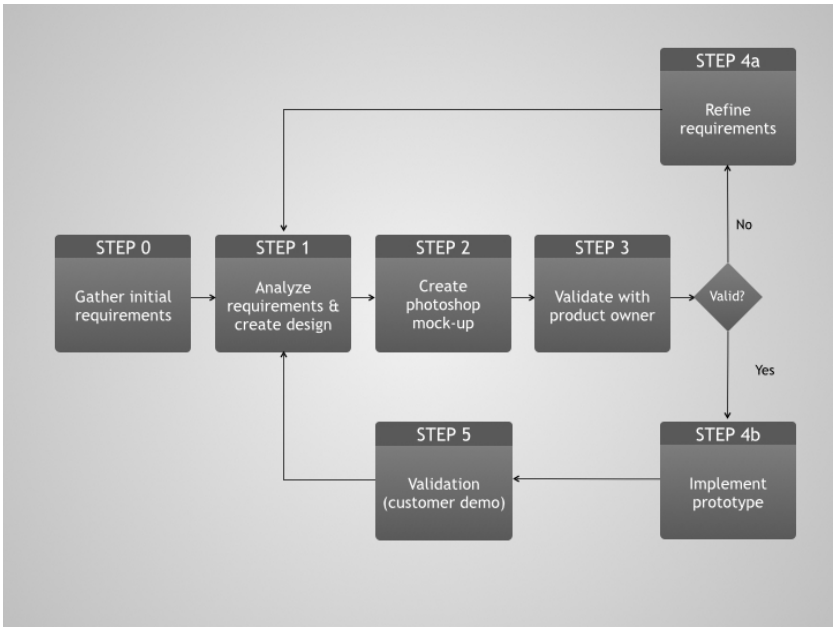

Figure 4.1: The process used for creating and improving our prototypes

or by the customer after a demo, depending on the situation. When most of the known requirements were implemented in the prototype, the product owner arranged a demo with a potential customer in order for the team to get feedback and new ideas for improvements for the next version. In addition, internal demos were also used informally to gain feedback from other developers and colleagues.

The process evolved throughout the project. For example in the very beginning, the process was much less established than in the end. For example creating the photoshop mock-up and validating it with the product owner only started after the mobile app prototype had already been completed, i.e. steps 2 and 3 were only used for the tablet version prototyping. This was mainly due to the fact that the graphic designer wasn't that involved in the project before the tablet prototyping started.

The prototyping was carried out over six months, and roughly half of the time was used on the mobile app version and the other half for the tablet version. During the six month period, we iterated through the process roughly 8-10 times, including two or three customer demos. In addition, the product owner showed the demo to other potential customers at conferences and other events, however that was outside the described process.

4.1.1 Step 0: Gather initial requirements

Some requirements were gathered already before the prototyping project began. This part of requirements engineering was carried out mainly by the product owner. Furthermore, the company had an employee who has previous worked on and implemented a similar system for a customer, and thus provided important insight into the critical aspects of such a system.

These initial requirements were at a very high level when the prototyping began. Examples of these initial requirements are

- Scheduling system should integrate with the payroll system and utilize the payroll system's data
- Scheduling system must be usable on mobile devices.
- Managers will use the tablet version of the app while employees will use it on their mobile phones
- App must be very fast and easy to use, because managers only have +/- 15 minutes available per day to use computers or tablets
- App must be available for Android and iOS platforms
- User can login with an NFC tag or with a PIN code

Furthermore, with the help of the employee with previous experience of such systems, we were able to identify and analyze three stakeholders at the customer site: the schedulers, employees and business managers.

4.1.2 Step 1: Analyze requirements and create design

The first step of our process was to analyze and refine the requirements and to create a UI design for the prototype based on the requirements. As an example, the first iteration for the tablet app prototype included the following requirements:

- Manager can quickly see open shifts for a location in the home page
- Manager needs a calendar view for the schedule
- Manager needs to see the schedule of a specific location
- Manager needs to see the schedule of a specific employee

Based on these requirements we came up with our UI design where the home page is the schedule view that satisfies all of the requirements. Open shifts are highlighted in the schedule with a different color so that they can be easily spotted. The schedule in the home page is presented as a weekly calendar view, similar to a timetable. The manager can toggle between two different tabs, where the first is grouped by location and the second is grouped by employee.

4.1.3 Step 2: Create photoshop mock-up

After having created a UI design to satisfy the requirements, we created a photoshop mock-up of the UI design before starting the implementation in code. This step was added to the process only after the mobile phone prototype had been completed and after the graphical designer joined the team. During the prototyping of the mobile phone app, it was noticed that the initial UI design and its implementation were often not accepted in validation by the product owner, resulting in a lot of rework with the code.

This happened mainly due to the developer misunderstanding the requirements or the product owner's original intention behind the requirements. These misunderstandings happened mainly due to two reasons. First, the requirements lacked details, were ambiguous and/or contradicting, and thus they were open for intepretation. And second, the developer didn't have as good an understanding of the big picture as the product owner.

Having noticed this pattern of misunderstandings, we realized that it is much faster and cheaper to visualize and iteratively improve the UI design in photoshop than in code (HTML/JS/CSS). We also learned that the clear visualization of the UI design resulted in the code implementation of the prototype becoming a lot easier and faster, since the details had already been decided and didn't need to be considered while coding. Therefore step two was added into our process: before the developer even touched the code, the UI design created in step 1 was visualized in photoshop by the graphical designer.

The photoshop mock-ups were created by the graphical designer, however the developer was also involved in the decision making process. The mock-ups were already iterated upon by the developer and graphical designer before moving on to step three, for example when coming up with a better idea while working on the original UI design. Often times the ambiguous and unclear requirements that were missed during step 1 were spotted while creating the mock-up, and thus we were able to consult with the product owner already in step 2, instead of during the code implementation. The mock-ups were used in our development process only for internal use, and not shown to the

customer.

4.1.4 Step 3: Validating the mock-up with the product owner

When the photoshop mock-up was complete, it was delivered to our product owner for validation. The developer and graphical designer described the UI design decisions that were made as well as their reasons, and in some cases alternatives were presented so that the product owner could choose which one was selected for implementation. Whenever possible, this was done in a meeting in the conference room, however sometimes we had to email the mock-ups and the description to the product owner.

During the prototyping of the tablet app, all the mock-ups in all the iterations were accepted and none of them were rejected. This was mainly due to close communication during step 2 between the entire team, which enabled us to notice all the major problems before the final validation in step 3. However, in a few iterations some minor details were changed during the review in step three.

4.1.5 Step 4

4.1.5.1 Step 4a: Refine requirements

If product owner rejects the proposed mock-up, it means there are fundamental in the UI design either due to big misunderstandings concerning some of the requirements or some new or upcoming requirements which would be prevented by the proposed UI design. As a result, these new or updated requirements need to be taken into account and we go back to step 1 with the refined requirements specifications.

4.1.5.2 Step 4b: Implement prototype

If the product owner validates the mock-up, it means that we are good to go and the prototype can be implemented in code. In this project, the developer first implemented a rough version of the software design: a simple UI and the related backend services. The rough version was first implemented to make sure the software design is technically feasible and can be implemented, i.e. making sure that the selected technologies or the existing architecture don't prevent the new implementation.

After the rough version was completed, the developer started fixing the user interface with the intention of creating the exact same result in code

as was in the mock-up. However, in some cases this was either not possible or would have been incredibly time consuming to achieve. Therefore, some compromises were made on occasion, mainly when the time saved was more valuable than getting precisely the same result as in the mock-up.

After the developer considered the implementation to be ready, a demo was shown to another team member or a colleague from outside of the project in order to gain some feedback, for example to get an outside opinion of the usability of the new features.

4.1.6 Step 5: Validation (customer demo)

After implementing the prototype it was reviewed within the team and occasionally with other colleagues. The purpose of this review was to validate that the implementation is aligned with the requirements as well as validating that the prototype is in line with the overall vision for the product. The developer showed a demo to the product owner, describing any possible differences between the implementation and the mock-up and the reasons for these changes. Since the UI design was already validated after the mock-up (step 3), this step 5 validation only concerned small details. However, in the first iterations before we used mock-ups, many big problems were discovered during the validation.

Having reviewed the prototype together with the team and/or other colleagues, a meeting was arranged with a potential customer for the product when necessary. Essentially the customer demo was arranged only when the prototype was considered ready, i.e. the main feature was implemented and there were no more unimplemented "must-have" requirements that we were aware of. In other cases, when we knew exactly what we need to do in the next iteration, the customer demo was skipped and we moved straight to step 1 with the next requirements. The main reason for this was that we didn't consider it important enough to show the work-in-progress to the customer, but absolutely needed to get the feedback for the features when they were ready.

The customer meetings were roughly 2 hours in duration, of which the first hour was reserved for the presentation and the second hour for discussion. Of the first hour, roughly 15 minutes was allocated for the prototype demo. The demo consisted of showing the implemented features, with the intention of giving the customer a basic understanding of the flow within the app. The idea was to get feedback for the prototype and to validate that the implementation was aligned with the customer's ideas and beliefs of the product. It is important to note, however, that not all of the users would be the customer. Instead, the majority of the users would be our customer's

customers, none of whom were represented in these meetings. Therefore not all stakeholders were present, although the customer had a very good understanding of their customers' practices and processes.

The discussion was very unstructured and informal, and the main concerns in the discussion were the customer's existing systems which this new product would eventually replace. The customer described in detail how certain features worked in their existing systems, and was able to identify e.g. which existing features were absolutely necessary and which features were implemented in such a way that didn't properly support their business processes. In addition, the customer gave examples of difficult cases they have had in the past that have resulted in lots of manual work, and was hopeful that this new product would be able to automatically handle at least the most common situations.

Based on the discussion in these meetings and the feedback received, we were able to discover new requirements as well as to get more detailed information and a better understanding about existing requirements. Furthermore, all members of the team were able to gain a much better big picture understanding in these discussions.

4.2 Understanding customer requirements

Understanding customer requirements was identified as a challenge during our project, which resulted in the development team misunderstanding some requirements. These misunderstandings happened for two reasons: lack of details in the requirements and not understanding the big picture.

Our team utilized three different approaches to resolve these issues, which helped us understand the requirements better. First, we used mock-ups in order to find problems with the UI design as early as possible. Second, we demoed the prototype to both the customer and colleagues to get outside feedback. Third, we had an expert colleague available, who had more knowledge about the domain. Through these three means, we were able to discover the lacking details in requirements and to gain better understanding of the big picture.

4.2.1 Mock-ups

Prototyping with the mock-ups helped the development team gain better knowledge and understanding of the customer requirements. We learned that using photoshop mock-ups before the implementation was an effective way to validate that the development team had understood the requirements

correctly. Mock-ups were only used for our internal purposes, and were not shown to the customer. The mock-up was like a proxy between the requirements engineering and development. By using it as a filter in our process. we were often able to identify the UI designs which weren't accurately aligned with the requirements. Thus we were able to avoid unnecessary coding, and instead we were able to bridge the knowledge gap between requirements engineering and development early on. Furthermore, we were able to identify the missing pieces of the UI design and resolve them relatively quickly. This was possible because a mock-up is a very detailed representation of the UI design, and therefore the problems were much more clearly visible in the mock-up than in a written requirement description.

One example of utilizing mock-ups to understand the requirements more clearly was in the beginning of the tablet version prototyping. We needed to implement the feature "Manager can view schedule in a calendar view". In our process, during Step 1: Analyze requirements and create design, we concluded the main purpose of the view is that the manager can get a quick overview of the schedule. One example use case was that s/he can make sure there are no shifts that haven't been assigned to an employee. Therefore, the shifts in the schedule will be highlighted based on the shift's status (green/yellow/red) to make it easy for the manager to spot the shifts that need urgent attention (red color). Furthermore, clicking on a shift in the schedule will open the shift's detailed view, where the manager can perform any necessary actions, such as assigning the shift to an employee, editing the shift's information or publishing it to the mini-market.

With this information we began to create the mock-up for the schedule view. While creating the mock-up, it became clear that there were details missing from the UI design that had to be resolved. The most fundamental issue was how to group the data in the schedule. It was obvious that the columns in the schedule would represent days in the schedule. However, the rows in the schedule could be organized in many ways. In addition, there could be hundreds of shifts per day, which means that the shifts need to be grouped somehow. After all, the main purpose of the view was that the manager could get a quick overview of the data, meaning that there cannot be hundreds of rows for each column.

Having created a few basic mock-ups, we had another meeting with the product owner to discuss and resolve these issues. We considered different use cases for the manager in the schedule view to try and decide how the grouping should be done. We concluded that no single view would satisfy all the different situations and decided instead to create three different tabs, each of which groups the data differently. In the location tab, the manager could select shifts for one location (e.g. a single restaurant), and the shifts would be

grouped by job profile (i.e. different rows for chefs, waiters and bartenders). In the employee tab, the manager could select one or more employees and see all of their shifts, grouped by location. And finally the period tab would show all the shifts in all locations for the selected time period, grouped by location, showing the total cost of all the shifts per location.

Eventually only the location tab was implemented in the prototype due to time constraints and other issues being of higher priority. Even so, everyone in the project team gained a clear understanding of how the schedule view worked, and we were able to fill in the lacking pieces and details of the original requirement. Furthermore, during the process we had to analyze and identify what the customer really needs, resulting in a UI design that was optimized to create value for the customer. In this way, using mock-ups in prototyping helped us understand the requirements as well as refine the requirements further.

4.2.2 Demo and feedback

The meetings and demo sessions with the customer were found to be extremely useful for everyone in the team. Furthermore, the development team also received constant feedback from the product owner, who had a much better big picture understanding. The product owner gained more knowledge by having regular meetings with potential customers. All of the following are ways which helped our team gain a better understanding of the customer's environment, processes and the requirements.

First of all, the team gained a better big picture understanding, e.g. by the customer explaining the problems they are having today in their business process and how their old existing system doesn't support specific parts of their processes. Secondly, we gained valuable feedback and validation for our UI design and implementation, as well as improvement suggestions. Thirdly, our team was able to run ideas by the customer and ask questions about upcoming requirements in advance. Fourth, we were able to challenge some ideas and get justification for some requirements that at first seemed out of place. Having understood the real reasons behind the requirements and thus improving our big picture understanding, we were actually able to improve and refine the requirements. And finally, the prototype was used in the discussion to make sure everyone in the session could follow which part of the solution was being discussed, minimizing the risk of communication misunderstandings with the customer.

On the other hand, the customer represented only a part of the potential users. The customer's customers were essentially unrepresented in the demo, and therefore we were unable to gain feedback and validation from them.

Therefore these demo sessions would have been even more useful for us if we were able to get feedback from all the stakeholders.

4.2.3 Expert colleague

Having an expert available with experience of implementing a similar system for another customer helped in the beginning of the project. The development team had a very little knowledge of the domain when the project started, but was brought up to speed during the prototyping with the help of the expert. Understanding customer requirements as a whole is very difficult without understanding the domain, because otherwise one cannot understand how different things and ideas are related to each other.

Furthermore, due to the past experience in the same field, the expert was able to describe the problems they faced in their project so that we could avoid the same pitfalls. Thus we were better able to understand the full impact and the risks of some requirements that might have otherwise come as a surprise later on. Therefore, having an experienced colleague helped the team understand the requirements better.

On the other hand, the customer in this project was very different from the project where the expert had worked. Therefore the requirements in the two project were also relatively different. As a result, the expert was mostly helpful for the high level concepts instead of detailed UI or software design decisions.

4.3 Defining ASRs

Prototyping helped us define architecturally significant requirements in two ways. First, we were able to understand, clarify and refine existing requirements, which enabled us to evaluate their architectural impact. And second, we were able to find new architecturally significant requirements, which might have otherwise stayed hidden and caused problems in the future if they required architectural changes. These were achieved both in customer demos and during the development process, in the implementation and the UI design. Furthermore, our prioritization and selection of requirements to prototype was heavily focused on ASRs, which enabled us to concentrate our efforts and thus helped us define the ASRs.

#	Requirement	Origin	Implemented
1	System must be able to serve multiple customers (multi-tenant environment)	Initial requirement	No
2	Scheduling system should integrate with the payroll system and utilize the payroll system's data	Initial requirement (refined during prototyping)	Partially
3	Scheduling system must be usable on Android and iOS mobile devices.	Initial requirement	Yes
4	Scheduling system can automatically assign employees to a schedule	Initial requirement	No
5	Managers will use the tablet version of the app while employees will use it on their mobile phones	Initial requirement	Yes
6	App must be very fast and easy to use, because managers only have +/- 15 minutes available per day to use computers or tablets	Initial requirement	Yes
7	User can login with an NFC tag or with a PIN code	Initial requirement	No
8	Shifts have a status which describes its phase in its life cycle	Initial requirement (refined during prototyping)	Yes
9	Shift can be released from employee (e.g. if employee is sick), notifying the manager to find a replacement	Initial requirement	Yes
10	Shifts are only visible to employees who have the correct job profile, skills and/or certificates	Initial requirement (refined during prototyping)	No
11	Shifts are only visible to the employees in the customer's employee pool	Prototyping / Customer feedback	No
12	Shifts consist of different activities that are time-tracked and calculated separately	Prototyping	No
13	System should send SMS-notifications to users when customer needs urgent reply	Prototyping / Customer feedback	No
14	Manager can see the real-time cost of the whole schedule	Prototyping	No
15	Manager can see the real-time potential cost of the shift when assigning an employee	Prototyping	No

Table 4.1: List of architecturally significant requirements

4.3.1 Clarifying ASRs

Table 4.1 lists all the architecturally significant requirements in the project. Out of the 10 initial requirements, three were refined during the prototyping: requirements 2, 8, and 10. Requirements 2 and 8 were refined by finding new requirements related to them. Furthermore, requirements 8 and 10 were refined further during implementation and customer demo by gaining better undertanding of the requirements as explained in section 4.2. Below is a detailed description of how each of these three requirements were refined.

Requirement 2 (Integration with payroll) was refined simply by finding the requirements 14 and 15, which are concrete ways of utilizing the integration to bring value for the customers.

Requirement 8 (shift status and life cycle) was also clarified during prototyping. It was further refined by requirement 12 as described in section 4.3.2.2. Furthermore, we were able to refine requirement 8 by iteratively trying to improve and optimize the statuses. Initially, there was a long list of shift statuses, which were changed and iterated upon multiple times during the prototyping. We were able to remove unnecessary statuses and combine some of them together in order to create a more logical flow for the shift life cycle. For example, at first we had separate statuses for the shift inside the schedule and for the shift inside the mini-market. However, with a few changes and decisions we were able to combine these into a single status that describes the shift's current state. Optimizing the statuses was very helpful overall, because over half of the features in the system were somehow related to the statuses or changing the shift's state. Therefore we were able to simplify the app by reducing the amount of controls required.

Requirement 10 (Shifts only visible to employees with correct job profile, skills and certificates) was refined by gaining more information about the requirement in the customer demo discussion, as described in section 4.3.2.1.

4.3.2 Finding new ASRs

As shown in Table 4.1, we had 15 ASRs in total. Out of those 15, ten were from the initial requirements gathering and five were found during prototyping. Out of the five that were found during prototyping, two were found during the discussion following the customer demo. The other three were also discussed and validated in the demo session with the customer, but had already been discovered before during our prototyping process.

4.3.2.1 ASRs found during customer demo

The new ASRs were found during the customer demo through discussion and by us understanding customer's needs and requirements. As an example, requirement 13 (SMS-notification) was discovered through requirement 9 (sick employee). It was discovered when discussing with the customer about requirement 9: what the context is like in a typical scenario and what actions each user takes. According to the customer, a typical scenario is that the person calls in sick in the morning, often 1-2 hours before the shift is about to start. Therefore the manager only has very little time to find a replacement for the shift, and thus emails or app notifications aren't fast and reliable enough. Therefore it was decided that SMS-notifications should be used for these urgent situations, and thus requirement 13 was discovered.

As another example, requirement 11 (Shifts visible only to employees in customer's employee pool) was found during the customer demo. Before the demo we already knew the first version of requirement 10 (Shifts visible only to employees with the correct job profile). The purpose of both requirements 10 and 11 was that there could be a very large amount of shifts in the system, and employees should only see the ones that are relevant to them. For example a waiter cannot be the head chef of a restaurant, so the waiter doesn't need to be able to apply to such shifts.

During the customer demo we discussed what sort of restrictions the customer would need and asked for more example cases to better understand the requirements. We were able to refine requirement 10, when the customer mentioned that job profile itself was not enough of a restriction for work shifts. As an example, some of the customer's customers would require specific certificates from all employees, such as a hygiene passport. These restrictions cannot be implemented simply by job profile, for example some waiters might have it and other might not. Therefore we needed to refine requirement 10 to also include restrictions based on skills and certificates.

In the same discussion we were also able to find requirement 11. The customer mentioned an employee pool concept that they used, which we had not thought about. After asking for more details and discussing it further, it turned out to be a new ASR. Employee pools are essentially a group of employees with a common job profile, skill, certificate or location. Each of the customer's customers can manage their own employee pools. This enables the customer's customers to make shifts visible only to a specific group of employees that they want to offer the shifts to. It can be used for example as a priority list, i.e. offer the shifts first to the employees that have worked there before and who the manager trusts. If not all shifts are filled from employees in the pool, the remaining shifts could then be made publicly

available to everyone later.

Another benefit of the employee pools is that the customer's customers can add employees to a "ban pool". This could happen for example if the customer's customer's manager gets complaints about a waiter, and the manager decides to never hire the same waiter again. In this case, the manager doesn't want the employee to come to the same location anymore, and this employee will never be able to see or apply to any of the shifts for this location or for the customer, depending on the ban pool settings.

4.3.2.2 ASRs found during prototyping

Requirement 12 (Shifts consist of different activities) was found during the prototyping. It was discovered when we were implementing the time-tracking feature of the prototype, where the employee can change the status of the shift (e.g. start, pause and complete the shift). The controls of these status changes were relatively complex due to there being so many different statuses. We prototyped a few iterations with different UI designs but none of them worked very well, so we had a brainstorming session about how we could make the feature simpler and easier for the users. In addition we considered different situations that the users would have. Our original idea had been that we only need two activities: working and taking a break. However, we realised that there can be cases where the employees have to report specifically what tasks or activities they were working on at what time, and their salary might be dependant upon it. For example a factory worker might need to report how many hours they spent on each assembly line, and each assembly line might have a different hourly wage.

The result of the brainstorming was that shifts can have more than two activities. Therefore we decided to significantly change the controls for changing the shift's status. Instead of separate buttons for e.g. starting, pausing, stopping and completing the shift, we decided to have only three buttons: start, change activity and stop. While the shift is in progress, the employee can change the activity based on what s/he is currently doing. The different activities in each shift can be configured when the shift is created, for example there could be three different activities for three assembly lines as well as an activity for taking a break. The system would track the time separately for all of the different activities, and the integration with the payroll system would be able to utilize this data and calculate the salary correctly according to the time spent on each activity.

Requirement 12 also introduced some changes to the shift statuses, which affected requirement 8 (shift status and life cycle). Essentially finding the new requirement 12 automatically helped us refine requirement 8, because

obsolete statuses were removed and the statuses were simplified.

4.3.3 Architectural impact

We found that there were three different types of architecturally significant requirements: core features, quality requirements and requirements related to the application environment. Below is an example requirement in each of these categories as well as descriptions of their architectural impact.

4.3.3.1 Core feature

One of the most core concepts within the scheduling system are the shifts and shift statuses. Basically all of the features in the app are somehow related to either the shift or changing its status. Therefore one of the most important core features is 8 (shift status and lifecycle).

The architectural impact of this requirement is that it is linked to all other functional requirements in the system. Therefore changing the statuses means that the flow within the app changes and it would impact several features. Furthermore, if we would not have analyzed and refined the requirement during prototyping, it would have created a big risk that the statuses were wrong and/or insufficient, which could break the UI or software design of other features.

4.3.3.2 Quality requirement

The most significant quality requirement in the project was requirement 6 (App should be fast and easy to use). This requirement is affected mainly by the usability of the app, but also performance is somewhat related. However, in our project the performance of the system was not an issue and did not have any significant effects.

The reason why this requirement was so important was the fact that the users don't have a lot of time to use on the app, so the usability must be so good that any task can be achieved in the very limited time that the users have.

The requirement didn't directly affect the architecture, but it had to always be considered during the UI design of all features. Therefore it was essentially a constraint in many UI design decisions, as it forced us to create simple UI designs so that users could find anything easily and quickly. If the requirement would not have been considered during prototyping, there would have been a risk that the UI designs would be too complicated and everything would have to be redesigned later on.

4.3.3.3 Application environment

One example of an ASR related to the application environment in this project was requirement 3 (App should be usable on Android and iOS mobile devices). This requirement is related to the application environment, as it describes the enviroment in which the system will run and how it will be used. The requirement was resolved by the decision of which technology should be used to build the multi-platform app. This decision is described in more detail in section 4.4.2.

The requirement has a very large impact on the architecture, as the device and the selected technology strongly affect how the app should be created. In addition, it creates some constraints on what is possible to implement. Furthermore, changing this requirement or any decisions related to it at a later stage would be extremely expensive, as the entire prototype is built upon this foundation.

4.3.4 Prioritization

Prioritization helped us define ASRs, because we were able to focus our prototyping effort mainly on ASRs. This was achieved by selecting requirements to prototype based on three factors. First and foremost was the uncertainty involved for the feature. We believed it was more important to prototype the features that we didn't have enough knowledge about, since learning and resolving unknown issues were important goals during prototyping. Therefore we preferred implementing features that we didn't have a clear vision about, instead of features where the details were already known and where it would be obvious how to implement. The second most important factor was the expected value created for the customer. This was because we wanted to validate our expectations and to learn as early as possible if our solution would be viable, instead of implementing features that were technically difficult but which the customer had no interest in. Furthermore we wanted to validate with the customer that our solution actually satisfies the requirement. The third factor was that we considered how relevant the feature is in the overall vision for the product. In other words, features that were considered to be core features were valued higher than other requirements.

These factors directed us towards implementing mostly architecturally significant requirements. According to the first factor explained above, we wanted to resolve issues that were unclear, and architecturally significant requirements often have uncertainty. As per the third factor, we preferred prototyping with the requirements that were related to the core modules and the requirements that have dependencies with other modules or components,

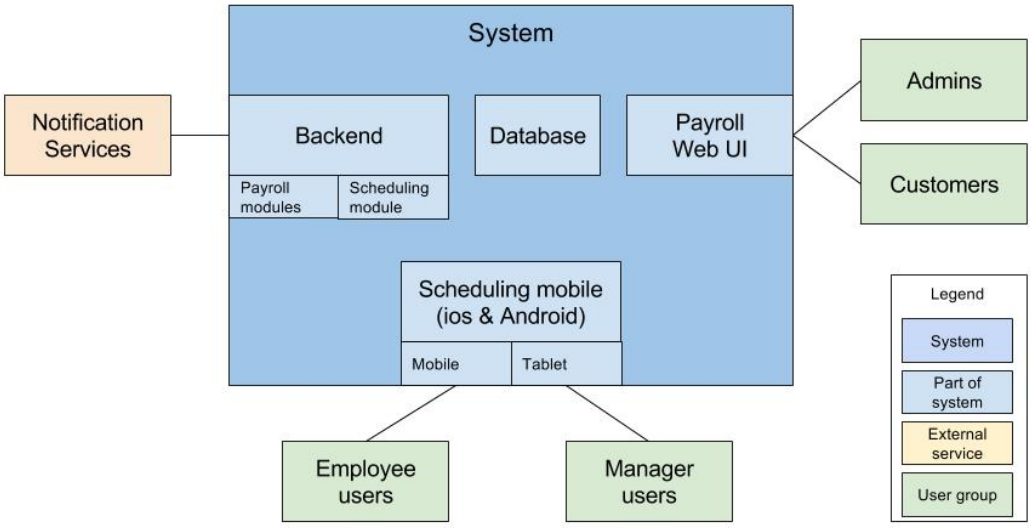

Figure 4.2: Context architecture diagram

which are usually ASRs as well. As a result, we ended up selecting and prototyping mostly ASRs in the project, which in turn helped us understand and define the ASRs more clearly. On the other hand, due to the second factor, we didn't prototype with all ASRs. This was because some of them were simply not significant enough in other ways, so we decided to postpone solving issues related to them because they were not of high enough priority.

On the other hand, some requirements were very significant and high priority based on our criteria, even though the requirements were not architecturally significant. These were mainly features that were absolutely necessary for users, e.g. a feature that an employee can see his/her own shifts. This is not architecturally significant in any way, but lacking such basic features would make it almost impossible to demo any real scenarios to the customer. Furthermore, quite a lot of effort was put into developing these basic features in order to satisfy and evaluate the quality requirements such as requirement 6 (app must be very fast and easy to use).

4.4 Software architecture design and evaluation

Figures 4.2 and 4.3 describe the software architecture of the created prototypes. The resulting software architecture was achieved through many

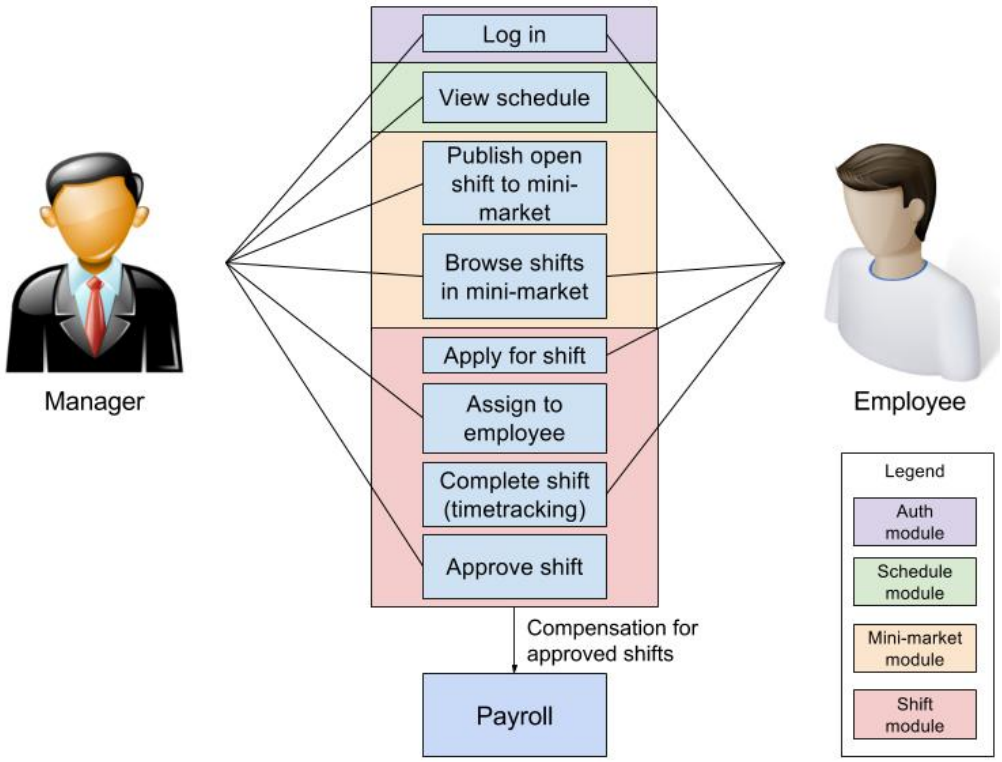

Figure 4.3: All four modules in Scheduling Mobile Client

architectural design decisions during the project. The most significant architectural design decisions were the following:

1. Which technology to use to implement multi-platform mobile client
2. Should scheduling backend be standalone or a module within payroll
3. How to separate mobile phone and tablet versions

Below is a description of how the final software architecture described in the Figures 4.2 and 4.3 was reached. After that, we describe each of the three architectural design decisions and how prototyping helped us solve the issues.

4.4.1 Factors affecting software architecture design

There were four factors that affected the software architecture. First of all, the architectural design decisions had a major impact on the architecture. They are described in the following sections. Secondly, the existing software architecture was effectively a constraint to the software architecture design. Thirdly, some new and refined ASRs caused changes in the software architecture. And finally, the software architecture of the mobile client evolved during evolutionary prototypes, as new features were implemented into new modules. Below is more detailed information about each of these factors.

The software architecture was very much affected by the existing payroll system. The payroll system's architecture was already established when the prototyping began and therefore we felt strongly encouraged to use the same technologies and make similar decisions whenever possible. As a result, many architectural design decisions were forced, such as using Scala as the programming language, using the same database as payroll and communicating with the client through a REST API in JSON format.

The software architecture evolved during the prototyping as new ASRs were defined. First of all, the requirement 13 in Table 4.1 meant that the system needs to integrate with external Notification Services. Secondly, requirements 10 and 11 affected the data model, since new concepts such as skills and certificates need to be introduced into the system as well as linking customers to employees to implement employee pools. Thirdly, the integration between scheduling and payroll was defined more clearly with requirements 14 and 15.

The software architecture of the mobile client was defined during prototyping. As new functionality was implemented, the new features were made into new modules. In the end of the prototyping, the app consisted of four

modules: the auth module, mini-market module, shift module and schedule module. The first three modules were for both the mobile and tablet versions, while the schedule module was only for tablet version. The features were somewhat split into modules based on the UI, since the navigation within the app was organized the same way as the modules. For example, when a manager opens the app the login screen is opened (auth module). After logging in, the schedule view is shown to the manager (schedule module). By clicking on shifts in the schedule, the manager can see the shift detailed view (shift module). And through the navigation bar at he top of the screen, the manager can move to the mini-market.

4.4.2 Technology to implement multi-platform mobile client

We used prototyping to learn more about two candidate technologies that could be used to implement the multi-platform mobile client. Two separate prototypes were created and compared before making the final decision. While creating the prototypes, certain key differences were noticed between the technologies, which helped choose the best alternative for our situation. Below, we describe the two technologies and their differences, as well as how we made the decision after prototyping with both technologies.

The first technology option was to create two native applications, one for Android and another app for iOS. The second option was to use the multi-platform Ionic framework, so only one app would be created. Originally the first option seemed slightly favorable, because usability of the product was considered extremely important and it is generally agreed that native applications simply have a slight edge over hybrid apps in terms of usability. However, native apps require a lot more work because two separate code bases are required, meaning that essentially all of the code has to be written twice (once for each platform). Although porting the code from one platform to another doesn't imply twice the effort, it still requires additional resources as well as more testing and maintenance work in the future.

In the beginning it was still unclear exactly what features will be included in this app. Therefore it was possible that some features require functionality that is not possible with Ionic framework, or that for some other (quality) reason the Ionic framework would prove to be insufficient. However, this risk was considered minimal and very unlikely. On the other hand, having created both the native and Ionic prototypes, it was very clear that the one created with Ionic was much faster to implement. First of all, the same code and components from the Payroll Web UI could be used, because it is also

written in AngularJS. Secondly, the integration with the backend was a lot easier, because the client communicates with the backend through the REST API in JSON format, and JSON is very much simpler to handle in javascript than it is in Java, Scala and Swift. Furthermore, the integration with the backend would have to be implemented twice on the two native apps. More testing effort would also be required with native apps, and maintainability would suffer since any changes would need to be made twice.

Therefore we ended up selecting Ionic instead of native, because Ionic had two significant benefits (improved maintainability and faster development) that were noticed and validated during prototyping, while the only downside was a risk that seemed very unlikely to happen.

4.4.3 Standalone vs combined with payroll

Prototyping was not used to decide between having a standalone scheduling backend and combining the backend with Payroll. However, it would have been possible to prototype with these two options by creating two prototypes and comparing them together, similarly as in Section 4.4.2 Technology to implement multi-platform mobile client. The reason why prototyping was not used was that the decision was relatively obvious and could be made without gathering more information. Below is a description of the two alternatives and how we made the decision.

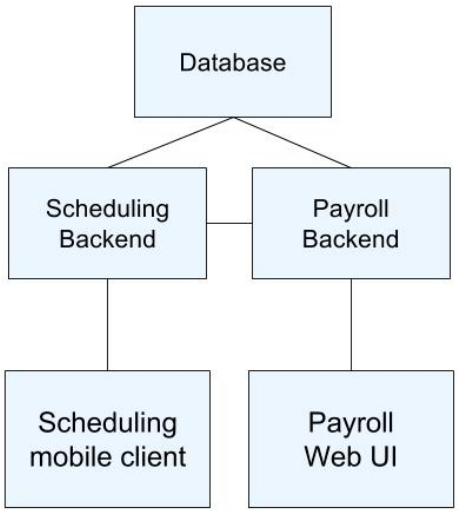

Figure 4.4: Standalone

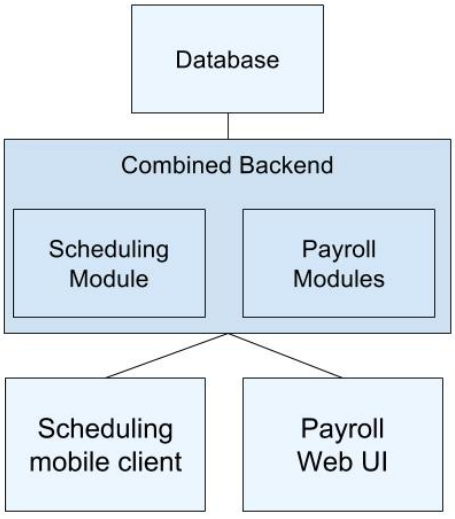

Figure 4.5: Combined with payroll

Figures 4.4 and 4.5 illustrate the architectural difference between the two options. One of the main benefits of creating a standalone product and thus

completely separating the scheduling backend from the payroll backend is that the scheduling could be sold, delivered and installed into environments with the payroll coming from a different vendor. In addition, the architecture might be more clear and logical, as the two products would be clearly separated from each other and the two backends could communicate with each other through their REST API.

However, one of the main advantages of the prototyped scheduling system is its integration with the payroll system, and having a completely standalone scheduling system is not what was planned. In addition, implementing integrations with different payroll vendors is simply not possible, so from a business point of view selling the products separately isn't feasible. In addition, separating the two servers from each other would introduce the need for at least some duplicate code, reducing the maintainability of the system. And finally the performance of the system would slightly suffer, since the mobile client would first call the scheduling backend through the REST API, which would then need to call the payroll REST API.

As a result, there were no tangible benefits in creating a standalone scheduling product, while combining the two had multiple advantages. Having discussed the two alternatives with the team in a meeting, we decided to combine the scheduling backend with payroll. Therefore scheduling was implemented as a module within the payroll server.

4.4.4 Separate mobile and tablet versions

Having two separate versions of the app was a challenge we faced which was solved through prototyping. We made a single prototype to figure out if combining the two versions into a single app was a viable option. With this exploratory prototype, we learned how this combination affects the software architecture and whether or not it is a viable option in regards to software maintainability. Below is a description of the problem, our options and how we came to our conclusion with the help of prototyping.

Initially we had two ideas how to solve the issue of having two separate versions of the app. The first option was simply to create two separate applications: one app for the mobile phone and another one for the tablet. However, based on the requirements it was clear that the two versions share many features, and therefore separating the apps into two projects would result in a lot of code duplication. Therefore it seemed like an interesting option to try and combine the two versions into the same app, but change the behaviour of the app based on the user's device.

We decided to prototype with the combined app as we considered it to potentially be the much better option. However we didn't know for sure how

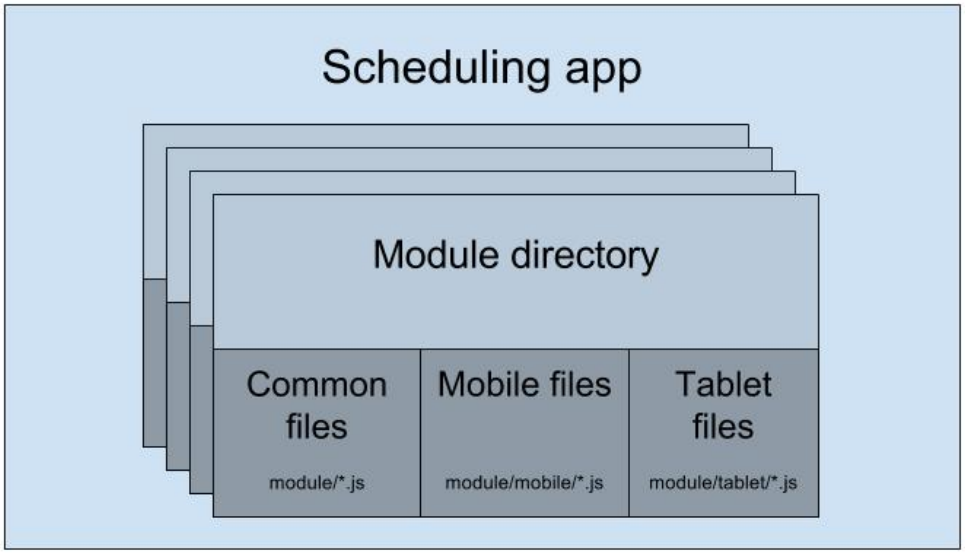

Figure 4.6: Scheduling app file structure

it would be implemented technically and what implications it would have on the architecture. During the prototyping, we learned that we could modify the behaviour of the app based on the device that is being used. This was achieved by conditionally loading specific HTML/JS files depending on the device. The two different versions were easily achieved, however it became very unclear and hard to see which features and which files were being used for each version. This increased the complexity of the app significantly, since not only did we have these two versions, but also two different platforms that can sometimes use different HTML/JS files.

Therefore we decided to improve the prototype further and to come up with a solution to reduce the complexity. This was eventually achieved by clearly defining which files are for which version. Figure 4.6 illustrates the concept of how the files were structured within the app to try to achieve clarity. The app is separated into four modules, described above in Figure 4.3. In each of the modules, the files common to both versions and platforms were placed directly inside the module directory. The version or platform specific files were then placed into subdirectories. For example if the login were the same for all versions and platforms, the login files would be found in auth/login.js and auth/login.html. On the other hand, if mobile and tablet versions had different authentication methods, different files could be found under auth/mobile/login.js and auth/tablet/login.js, and the correct one would be automatically loaded depending on the user's device.

This structure helped with achieving sufficient clarity. In addition, after the prototyping we noticed that there were only three files that are written separately for mobile and tablet versions, and there were less than 10 files that existed only for one version or the other. Therefore the additional complexity was considered insignificant enough that the combination of the two versions into one app was a better choice due to improved maintainability than creating two separate apps. In this way, prototyping helped us decide that combining the two versions into a single app was the best option.

Chapter 5

Discussion

In this study we used three different kinds of prototypes for different purposes. Firstly, the project itself was a prototype to evaluate the viability of our solution and to gain feedback from potential customers. Second, we used exploratory and evolutionary architectural prototypes to support the design and evaluation of the software architecture. And thirdly we used UI mock-ups to support the design and implementation of visible features.

In sections 5.1, 5.2 and 5.3 we discuss how these prototypes supported understanding customer requirements, defining ASRs and the design and evaluation of the architecture. In addition, we compare our results with existing literature.

5.1 Prototyping to understand customer requirements

Prototyping helped us understand customer requirements in two ways: using mock-ups and showing demos to customer to get feedback. These methods helped us solve the two main reasons for misunderstanding customer requirements: lack of details and not understanding the big picture. In this study, we also had an expert colleague who helped us understand customer requirements, however this was not related to prototyping.

During our development process, we created UI mock-ups with photoshop for internal purposes, i.e. they were not shown to the customer. The mock-ups helped us understand customer requirements better by filling in the lacking details of the requirements in two ways. First, we were able to validate that all team members had understood the requirement the same way, minimizing the risk of customer requirements being misunderstood. Second, the mock-ups were very detailed representations of the UI design. This means

that the mock-ups could not be created without making very many decisions regarding the details and we therefore no longer lacked the details. In short, using mock-ups minimized the risk of misunderstandings and made sure the requirements and the UI design contained all the crucial details required. On the other hand, mock-ups did not help with understanding the big picture.

Customer demos and feedback helped us understand the big picture. In the customer demo sessions, the newest version of the prototype was presented to the customer, followed by discussion about e.g. the implementation and what the next prototype would include. These sessions helped us understand the big picture better by having the customer describe the problems they are facing in their business process. Furthermore, these sessions gave our team an opportunity to ask questions and challenge some requirements in order to understand the real reason behind some requirements. Moreover, we were able to further validate that the requirements had been correctly understood and that our proposed solution was aligned with the customer expectations. In addition to the demo sessions, our product owner also had several meetings with potential customers, gaining a better understanding of the different requirements of different potential customers. In both the demo sessions and meetings, the discussion was strongly supported by having a functional prototype available, as one could pinpoint in the prototype exactly which feature was being discussed. This made communicating with the customer easier and ensured that mutual understanding was achieved.

Almost exactly the same results have also been discovered in previously published work (see section 3.5.2). Windsor and Storrs found that prototyping can reduce the risk of misunderstandings and different interpretations of system specifications and that prototyping enables the gathering of early feedback [37]. Furthermore, it was found that prototyping is a way to improve the communication between stakeholders [7, 21, 26], and that prototyping helps in reaching mutual understanding faster [21].

However, we also found that customer demos helped us understand the big picture, which is conflicting with previous work. Käpyaho and Kauppinen found that prototyping did not help with understanding the big picture [21]. One possible explanation for the difference is that we had a strong focus on ASRs and software architecture during our study and during the customer demos. This meant that when requirements were discussed in customer demos, we were already considering their architectural impact. Furthermore, as was discussed in section 3.4, the main benefits of integrating RE and SAD include better understanding of system specifications and the ability to create specifications that can be economically implemented. This suggests that in our project we were somewhat successful in integrating RE and SAD together, as we were able to realize these benefits. Therefore it is possible that

understanding the big picture was not a direct result of prototyping, but was achieved through integration of RE and SAD and by focusing on ASRs and software architecture.

5.2 Prototyping to define ASRs

Prototyping was used to define ASRs in two ways during the project. First, prototyping enabled us to understand, clarify and refine the ASRs, so that we could evaluate their architectural impact. Secondly, prototyping helped us find new ASRs, minimizing the risk of the ASRs staying hidden and causing big architectural changes in the future. Both clarifying and finding new ASRs were achieved during the implementation of the prototypes as well as in customer demos, and neither was supported by UI mock-ups.

Some ASRs were clarified and refined by iteratively trying out different options. During each iteration, we learned something new about that particular feature or requirement. This enabled us to create a better design for it, as we understood the underlying requirement and business need better.

Other ASRs could be defined in more detail after discovering new details about the requirement in the discussion following the customer demo. This happened mainly after we had implemented the first version of the prototype, and during discussion with customer we learned why it was not sufficient.

New requirements were often found in the discussion following the customer demos. During the discussion we wrote these new requirements down, and the majority of them turned out to be architecturally significant requirements. In addition, some ASRs were found hidden inside other requirements, or they could be read between the lines in discussions with customers. They were found partly because we were focusing on ASRs. When customer mentioned something that sounded like it could potentially be an ASR, it was spotted and we asked for more details about it.

New requirements were also found during the implementation of the prototypes. We sometimes noticed that the design was not optimal during the UI design or implementation phase. If it simply could not be iteratively improved to an acceptable quality level we decided to scrap it. Then, during a brainstorming session we were able to come up with new ideas, designs and requirements, which would satisfy the customer's quality expectations.

We were not able to find any existing literature on prototyping in the context of defining ASRs, meaning that these results are rather new. However, our results are somewhat supported by Bardram et al., who found that architectural prototyping helps in reducing architectural risk [3]. Architectural risk was reduced in this case by discovering hidden ASRs.

5.3 Prototyping to design and evaluate software architecture

Prototyping was used in software architecture design and evaluation in two ways. First was creating two alternative exploratory architectural prototypes, evaluating them, comparing them with each other and selecting the better option. Second was creating a single experimental architectural prototype to evaluate the architectural impact, after which we had more information to make the architectural decisions. In our project, prototyping was used in two of the three most important architectural decisions, and could have been used in the third one as well if the decision had been more difficult.

Software architecture design was an iterative process, which was driven by architectural design decisions. These design decisions were analyzed, and based on the situation we created either exploratory prototypes, an experimental prototype, or no prototypes at all. Where prototyping was used, the implemented prototypes were evaluated to understand the architectural impact, before confirming the final architectural decision.

Our software architecture design process followed the three activities of SAD presented by Hofmeister et al. [19] (see section 3.2). Architectural analysis was strongly supported by prototyping, as it helped us refine and find new ASRs. We also found architectural analysis to be affected by constraints of the existing architecture. This is also supported by Chen, who found that the constraints imposed by the existing architecture are indicators for ASRs [8]. During architectural synthesis we implemented exploratory, experimental or evolutionary architectural prototypes to make architectural design decisions. And finally, during architectural evaluation the candidate architectures were evaluated against ASRs and then validated or invalidated.

Chapter 6

Conclusions

Prototyping to define ASRs encourages requirements engineers and software architects to work together. Prototyping helps requirements engineers to discover and understand customer requirements, while software architects can create prototypes to evaluate the architectural impact of these requirements. Software architects can then provide feedback to requirements engineers, enabling informed requirements decisions and improved requirements negotiation with customers.

Requirements engineers and software architects should collaborate to prepare relevant prototypes for customer demos. Both requirements engineers and software architects can gain valuable feedback and a better big picture understanding from the discussion during the customer demos. UI prototypes support this discussion by helping avoid misunderstandings as well as reaching mutual understanding faster. Architectural prototypes can be used to demonstrate and validate the quality requirements.

Further studies are needed to evaluate the findings of this thesis. More studies are required to verify how prototyping should be used to better understand the big picture, as there are conflicting results in previous work. More research is also needed on the use of prototyping to define ASRs, as we could not find any existing literature on this topic. In particular, it would be interesting to explore how defining ASRs by prototyping supports understanding the big picture and the customer's business needs and processes.

Bibliography

[1] Andriole, S. J. Fast, cheap requirements prototype, or else! *IEEE Software 11*, 2 (March 1994), 85–87.

[2] Anish, P. R., Balasubramaniam, B., Cleland-Huang, J., Wieringa, R., Daneva, M., and Ghaisas, S. Identifying architecturally significant functional requirements. In *Proceedings of the Fifth International Workshop on Twin Peaks of Requirements and Architecture* (Piscataway, NJ, USA, 2015), TwinPeaks '15, IEEE Press, pp. 3–8.

[3] Bardram, J. E., Christensen, H. B., and Hansen, K. M. Architectural prototyping: An approach for grounding architectural design and learning. In *In Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004* (2004), IEEE Computer Society, pp. 15–24.

[4] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] Bäumer, D., Bischofberger, W. R., Lichter, H., and Züllighoven, H. User interface prototyping—concepts, tools, and experience. In *Proceedings of the 18th International Conference on Software Engineering* (Washington, DC, USA, 1996), ICSE '96, IEEE Computer Society, pp. 532–541.

[6] Boehm, B. W. A spiral model of software development and enhancement. *Computer 21*, 5 (May 1988), 61–72.

[7] Cao, L., and Ramesh, B. Agile requirements engineering practices: An empirical study. *Software, IEEE 25*, 1 (Jan 2008), 60–67.

[8] Chen, L., Ali Babar, M., and Nuseibeh, B. Characterizing architecturally significant requirements. *Software, IEEE 30*, 2 (March 2013), 38–45.

[9] Christensen, H. B., and Hansen, K. M. An empirical investigation of architectural prototyping. *J. Syst. Softw. 83*, 1 (Jan. 2010), 133–142.

[10] Cleland-Huang, J., Hanmer, R. S., Supakkul, S., and Mirakhorli, M. The twin peaks of requirements and architecture. *IEEE Softw. 30*, 2 (Mar. 2013), 24–29.

[11] Clements, P., and Bass, L. Relating business goals to architecturally significant requirements for software systems. Tech. Rep. CMU/SEI-2010-TN-018, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.

[12] Clements, P., Kazman, R., and Klein, M. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2001.

[13] Damian, D., and Chisan, J. An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management. *Software Engineering, IEEE Transactions on 32*, 7 (July 2006), 433–453.

[14] Ferrari, R., Madhavji, N., Sudmann, O., Henke, C., Geisler, J., and Schafer, W. Requirements engineering decisions in the context of an existing architecture: A case study of a prototypical project. In *Requirements Engineering Conference (RE), 2010 18th IEEE International* (Sept 2010), pp. 79–88.

[15] Green, J., Marnewick, A., and Pretorius, J. Requirements degradation for the creation of a first prototype. In *Management of Engineering and Technology (PICMET), 2015 Portland International Conference on* (Aug 2015), pp. 1777–1786.

[16] Guo, J. An approach for modeling and designing software architecture. In *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the* (April 2003), pp. 89–97.

[17] Heaton, N. What's wrong with the user interface: how rapid prototyping can help. In *Software Prototyping and Evolutionary Development, IEE Colloquium on* (Nov 1992), pp. 7/1–7/5.

[18] Hickey, A. M., and Davis, A. M. A unified model of requirements elicitation. *J. Manage. Inf. Syst. 20*, 4 (Mar. 2004), 65–84.

[19] Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., and America, P. A general model of software architecture design derived from five industrial approaches. *J. Syst. Softw. 80*, 1 (Jan. 2007), 106–126.

[20] Hull, E., Jackson, K., and Dick, J. *Requirements Engineering.* Springer London, 2010.

[21] Kapyaho, M., and Kauppinen, M. Agile requirements engineering with prototyping: A case study. In *Requirements Engineering Conference (RE), 2015 IEEE 23rd International* (Aug 2015), pp. 334–343.

[22] Lucassen, G., Dalpiaz, F., van der Werf, J. M., and Brinkkemper, S. Bridging the twin peaks: The case of the software industry. In *Proceedings of the Fifth International Workshop on Twin Peaks of Requirements and Architecture* (Piscataway, NJ, USA, 2015), TwinPeaks '15, IEEE Press, pp. 24–28.

[23] Maiden, N., and Rugg, G. Acre: selecting methods for requirements acquisition. *Software Engineering Journal 11*, 3 (May 1996), 183–192.

[24] Medvidovic, N., and Taylor, R. Software architecture: foundations, theory, and practice. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on* (May 2010), vol. 2, pp. 471–472.

[25] Nuseibeh, B. Weaving together requirements and architectures. *Computer 34*, 3 (Mar. 2001), 115–117.

[26] Paetsch, F., Eberlein, A., and Maurer, F. Requirements engineering and agile software development. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on* (June 2003), pp. 308–313.

[27] Robson, C. *Real World Research: A Resource for Social Scientists and Practitioner-Researchers.* Regional Surveys of the World Series. Wiley, 2002.

[28] Runeson, P., and Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg. 14*, 2 (Apr. 2009), 131–164.

[29] Schrage, M. Never go to a client meeting without a prototype. *IEEE Softw. 21*, 2 (Mar. 2004), 42–45.

[30] Seaman, C. B. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering 25*, 4 (Jul 1999), 557–572.

[31] Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., and Zelesnik, G. Abstractions for software architecture and tools to support them. *Software Engineering, IEEE Transactions on 21*, 4 (Apr 1995), 314–335.

[32] Shekaran, C., Garlan, D., Jackson, M., Mead, N. R., Potts, C., and Reubenstein, H. B. The role of software architecture in requirements engineering. In *Requirements Engineering, 1994., Proceedings of the First International Conference on* (Apr 1994), pp. 239–245.

[33] Sommerville, I., and Kotonya, G. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[34] Szekely, P. User interface prototyping: Tools and techniques. Tech. rep., In Proceedings of INTERCHI'93, 1994.

[35] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[36] Waldmann, B. There's never enough time: Doing requirements under resource constraints, and what requirements engineering can learn from agile development. In *Proceedings of the 2011 IEEE 19th International Requirements Engineering Conference* (Washington, DC, USA, 2011), RE '11, IEEE Computer Society, pp. 301–305.

[37] Windsor, P., and Storrs, G. Prototyping user interfaces. In *Software Prototyping and Evolutionary Development, IEE Colloquium on* (Nov 1992), pp. 4/1–414.

[38] Zhang, J., Chang, C., and Chung, J.-Y. Mockup-driven fast-prototyping methodology for web requirements engineering. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International* (Nov 2003), pp. 263–268.